ARL-TR-7579 ● JAN 2016

US Army Research Laboratory

Network Science Research Laboratory (NSRL) Discrete Event Toolkit

by Theron Trout and Andrew J Toth

ARL-TR-7579 ● JAN 2016

US Army Research Laboratory

Network Science Research Laboratory (NSRL) Discrete Event Toolkit

by Theron Trout and Andrew J Toth
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**			
1. REPORT DATE *(DD-MM-YYYY)* January 2016	**2. REPORT TYPE** Final		**3. DATES COVERED (From - To)** 10/2014–09/2105
4. TITLE AND SUBTITLE Network Science Research Laboratory (NSRL) Discrete Event Toolkit			**5a. CONTRACT NUMBER**
			5b. GRANT NUMBER
			5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Theron Trout and Andrew J Toth			**5d. PROJECT NUMBER**
			5e. TASK NUMBER
			5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) US Army Research Laboratory ATTN: RDRL-CIN-T 2800 Powder Mill Road Adelphi, MD 20783-1138			**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-7579
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			**10. SPONSOR/MONITOR'S ACRONYM(S)**
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			
13. SUPPLEMENTARY NOTES			
14. ABSTRACT Discrete event simulation (DES) enables the analysis of well-defined systems by evaluating the results of events acting upon the system. DES usually skips over timespans where no events occur to enable faster-than-real-time simulation. While this is a powerful and often needed capability, in the domain of real-time emulation, where the events need to be synchronized with other emulated and/or real components, time skipping is not feasible. This report describes a toolkit created by US Army Research Laboratory's (ARL) Network Science Research Laboratory (NSRL) for building a DES for injecting discrete events into real-time emulations in a controlled and reproducible manner. The resultant system, the NSRL Discrete Event Toolkit (NDET), has enabled high reproducibility of network emulations experiments for NSRL.			
15. SUBJECT TERMS Experiment Control, Simulation			

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Andrew J Toth
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UU	18	**19b. TELEPHONE NUMBER (Include area code)** 301-394-2746

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures iv

1. Overview 1

2. Architecture Overview 2

3. API Class Structure 3

3.1 Experiment_Driver 3

3.2 Discrete_Event_Base 5

3.3 Event_Logger_Base 6

4. Discrete Event Simulation Lifecycle 7

5. Performance and Thread-Safety Considerations 8

6. Planned Future Capabilities 8

7. Conclusion 9

8. References 10

List of Symbols, Abbreviations, and Acronyms 11

Distribution List 12

List of Figures

Fig. 1 NSRL 1

Fig. 2 High-level system architecture of NDET 3

Fig. 3 A unified modeling language (UML) diagram of the *experiment_driver* class 4

Fig. 4 A UML diagram of the *discrete_event_base* class 6

Fig. 5 A UML diagram of the *event_logger_base* class 7

1. Overview

The US Army Research Laboratory’s (ARL) Network Science Research Laboratory (NSRL) is composed of a suite of hardware and software that models the operation of mobile networked device radio frequency (RF) links through emulation (not merely simulation) (Fig. 1). NSRL enables experimental validation or falsification of theoretical models, and characterization of protocols and algorithms for mobile wireless networks. It is used for a range of experiments, from assessing in-network aggregation of network information for detecting cyber threats, to characterizing the impact of communications disruption on perceived trust and quality of information metrics delivered to Soldiers in tactical mobile environments. Unlike other experimentation facilities for research in wireless networks, NSRL is focused on Army-unique requirements like hybrid networks and extensive modeling of ground and urban effects on communications. NSRL supports investigation of traditional wireless networking challenges as well as more general network science research issues. NSRL's emulation environment is result of collaborative efforts between ARL and the US Naval Research Laboratory (NRL).

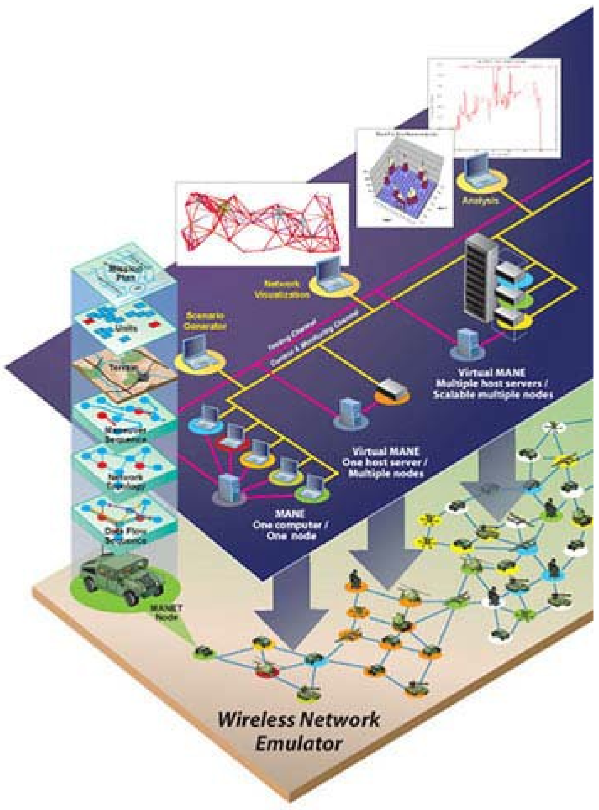

Fig. 1 NSRL

The primary emulation tools used by ARL are the Extendable Mobile Ad hoc Network Emulator (EMANE)[1] and the Common Open Research Emulator (CORE).[2] Researchers at NSRL developed the NSRL Discrete Event Toolkit (NDET) to improve the process of network science experimentation by providing fine control over timing of modeled events external to the experiment. Emulated networks developed by NSRL can model connectivity to hardware and software systems that are unaware that they are not operating on real networks. NDET is a lightweight, C++ application programming interface (API) for developing real-time discrete event simulation (DES) systems, which simulate inputs, outputs, or services required by the experiment.

The software and systems running on the emulated networks execute in real time unlike simulations, which typically execute faster than real time by jumping from event to event skipping the time between events. One of the biggest advantages of emulated environments is that the characteristics of the network and its behaviors can be fully controlled and are repeatable. This is particularly interesting when emulating wireless networks as reproducibility of experimental environments using real radios is often difficult as temperature and humidity changes, differences in seasonal foliage, and other factors can alter the performance of the wireless networks.

In concert with this consistency of network characteristics is the need for reproducibility of other experiment events. NDET has facilitated the development of experiment control and orchestration tools to provide such capabilities.

2. Architecture Overview

Figure 2 depicts the high-level architecture of the NDET components. Central to the operation of a simulation built using NDET is a dedicated controller that manages the emulation system. The controller's primary tasks are to determine events to be performed, launch execution of those events, log results, and update performance metrics. Events to be performed are stored in the discrete event definition table (DEDT) and accessed by the controller. Events in the DEDT essentially define the overall scenario to be executed. Each event is associated with a user-defined event execution class (EEC). A sample execution class would be one that executes UNIX commands. Such a class knows how to accept a valid command string and pass it to the underlying operating system. An instance of the EEC is passed to a worker thread in the worker thread pool for execution. The number of worker threads is specified by the user application using the NDET API and is bound only by the lightweight process limit of the system on which the emulated experiment is executed.

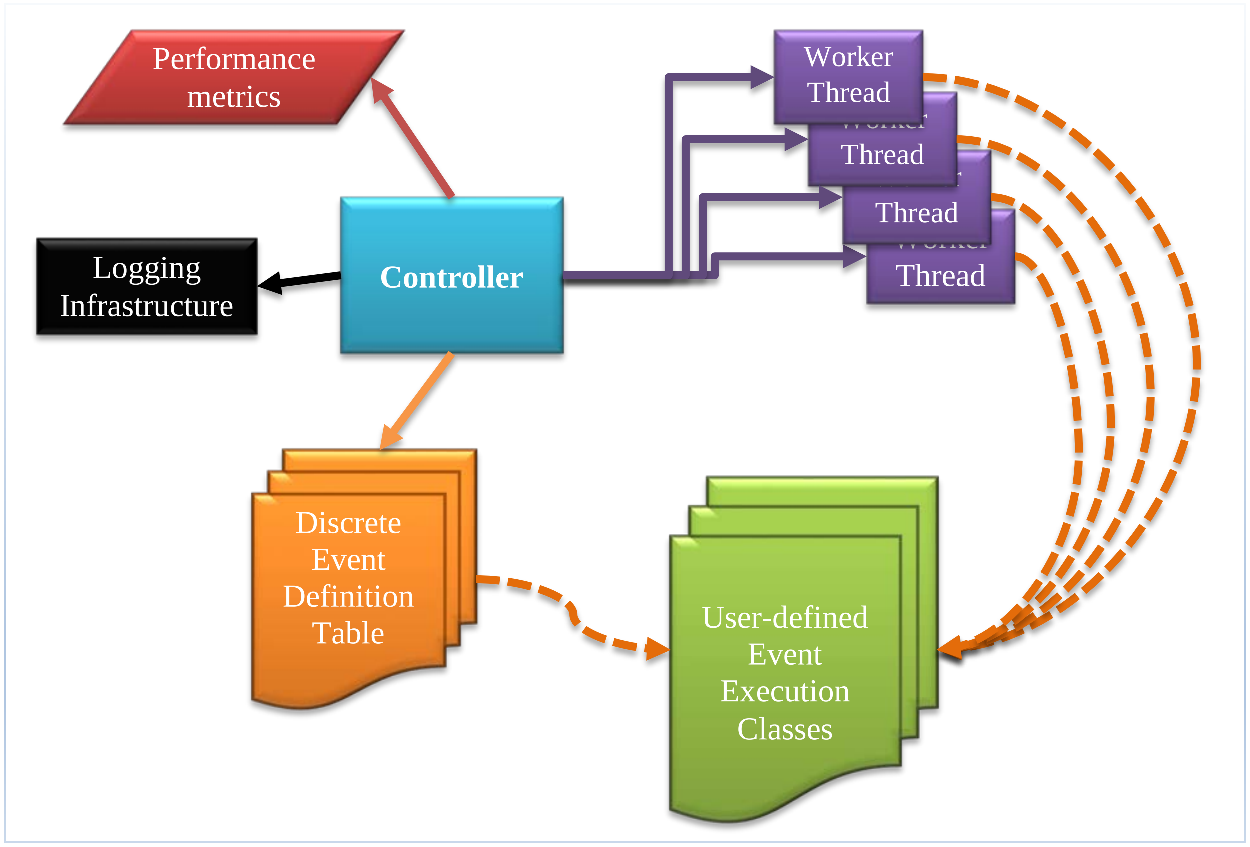

Fig. 2 High-level system architecture of NDET

3. API Class Structure

The toolkit's primary elements consist of 3 core classes, which define the totality of the API. These core classes are the *experiment_driver*, *discrete_event_base*, and *event_logger_base*. These are described in detail in the following sections.

3.1 Experiment_Driver

The *experiment_driver* implements the controller depicted in Fig. 2. Its structure is depicted in Fig. 3.

Controller
Discrete Event Definition Table
Performance metrics

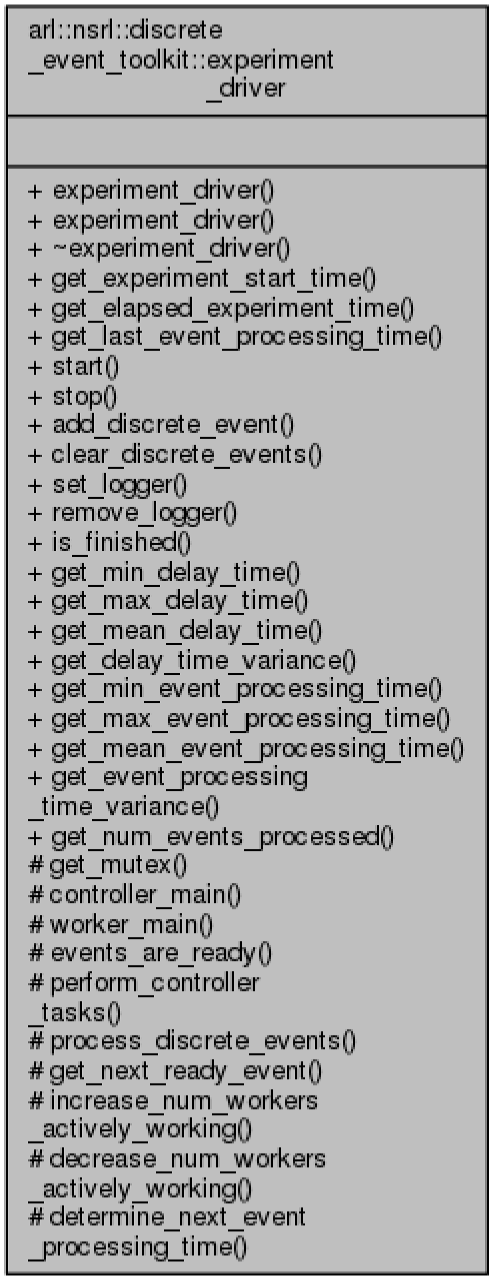
arl::nsrl::discrete_event_toolkit::experiment_driver
+ experiment_driver()
+ experiment_driver()
+ ~experiment_driver()
+ get_experiment_start_time()
+ get_elapsed_experiment_time()
+ get_last_event_processing_time()
+ start()
+ stop()
+ add_discrete_event()
+ clear_discrete_events()
+ set_logger()
+ remove_logger()
+ is_finished()
+ get_min_delay_time()
+ get_max_delay_time()
+ get_mean_delay_time()
+ get_delay_time_variance()
+ get_min_event_processing_time()
+ get_max_event_processing_time()
+ get_mean_event_processing_time()
+ get_event_processing_time_variance()
+ get_num_events_processed()
# get_mutex()
# controller_main()
# worker_main()
# events_are_ready()
# perform_controller_tasks()
# process_discrete_events()
# get_next_ready_event()
# increase_num_workers_actively_working()
# decrease_num_workers_actively_working()
# determine_next_event_processing_time()

Fig. 3 A unified modeling language (UML) diagram of the *experiment_driver* class

When the user instantiates the *experiment_driver*, the desired number of worker threads is specified. Next, EEC instances are provided and stored in the DEDT.

To initiate the experiment, the *start()* method is invoked. The invocation of this method marks the epoch time for the experiment. All discrete event times are specified relative to this time, which advances in real time in step with the system microsecond clock.

Upon reaching the specified start times for each event in the DEDT, the events are handed off to worker threads for execution. The driver tracks the time between the requested event start time and the time the event is invoked. These 2 numbers together provide a metric for measuring the accuracy of the NDET system in executing discrete events at the requested time. Casual observations from past usage have shown delays often to be in the single digit to tens of microseconds. Heavier system loads can introduce considerable jitter, however, and care should be taken when accuracy is most critical.

Performance metrics maintained by the controller include details on the delay time (i.e., delta between scheduled and actual event time). Users may query the controller for the minimum, maximum, mean, and variance of delay times for the current simulation run.

Metrics on event processing time are also maintained. This measures the amount of time spent running the *execute()* methods of EECs. Like the delay time, available metrics include minimum, maximum, mean, and variance of event processing times.

Finally, the controller maintains a count of the total number of events processed.

3.2 Discrete_Event_Base

The *discrete_event_base* is the base class for all user-defined events (Fig. 4). It provides the interface contract leveraged by the controller and worker threads to execute the events.

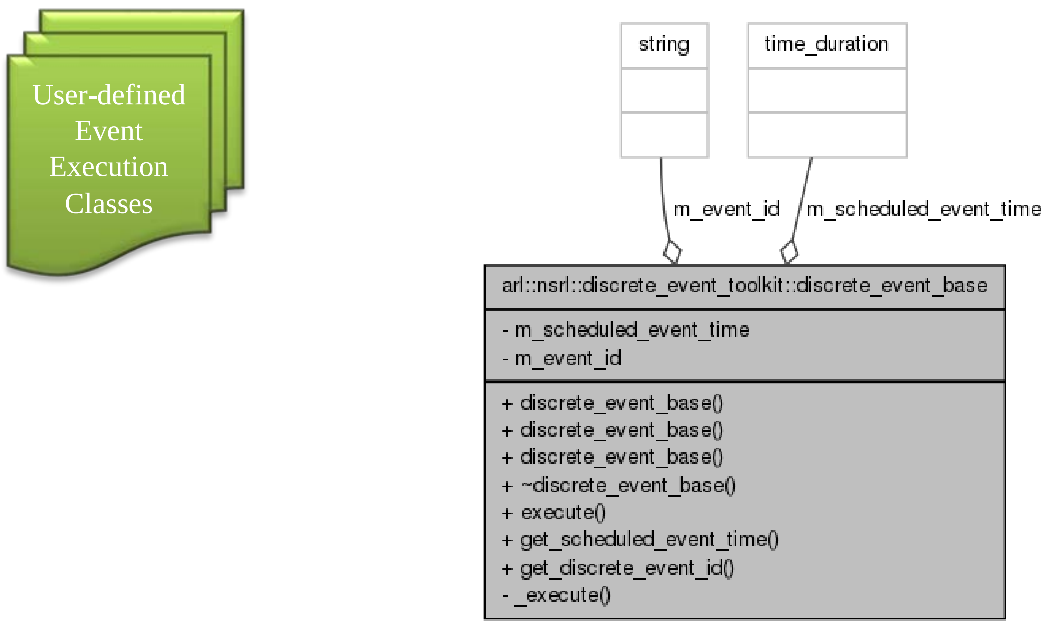

Fig. 4 A UML diagram of the *discrete_event_base* class

As long as no pointers, references, or handles are shared between *discrete_event_base*-derived classes, there should be no thread safety requirements placed on API users. If, however, this is not feasible, the developer will need to make appropriate usage of semaphores, etc., to ensure thread safe implementation. The *execute()* method can be thought of as the *main()* class of the discrete event. The developer may leverage any C/C++ capabilities to implement the desired behavior of the event. It is worth stating explicitly that the worker thread running the discrete event instance will be dedicated to running the given event until the *execute()* method finishes. Developers can exhaust the thread pool if their implementations perform long-running operations. In such cases, it would be advisable to fork or launch a new thread to perform these operations.

The *execute()* method will be called with 2 parameters. The first is the scheduled time when the event was to fire. The second is the delta time between the scheduled time and the time the *execute()* method was called. These data can be used to fine tune calculations or take other steps to mitigate timing errors.

3.3 Event_Logger_Base

The *event_logger_base* class (Fig. 5) provides the hook through which users may access the output of the NDET logging infrastructure. API users can provide an instance of a derived class to the controller and then collect and/or present the event log details to their users in their preferred format.

Logging Infrastructure

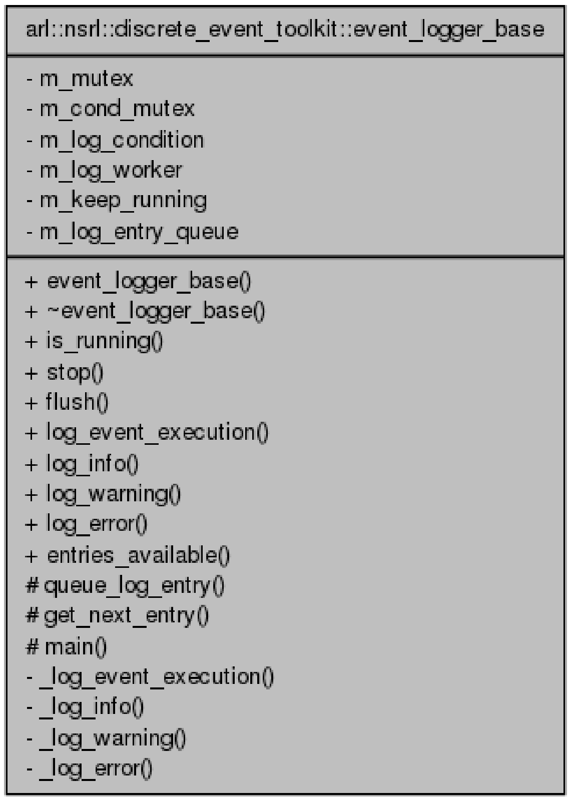

Fig. 5 A UML diagram of the *event_logger_base* class

Two default logger implementations are provided for convenience. These are *console_logger* and *syslog_logger*. The former sends all log output to standard out/standard error, as appropriate. The latter sends the log data to the syslog hander. The syslog handler is part of the GNU C library that opens a connection to the syslog daemon. When using the syslog handler, it is the responsibility of the API user to initialize syslog with an appropriate call to *openlog()*.

4. Discrete Event Simulation Lifecycle

There are 3 phases to the discrete event simulation lifecycle:

- setup
- execution
- post-run analysis

During the setup phase, instances of custom event objects are created and passed to the controller.

Once ready, the execution phase is initiated. The system launches the requested number of threads and invokes each event per the experiment schedule defined in the DEDT.

After the experiment finishes, the post-run analysis phase provides an opportunity for the user application to evaluate the run statistics collected and handle any custom experiment data collected during the run.

The system can then rerun the experiment as-is; modify and relaunch; create and run a new experiment; or shut down.

5. Performance and Thread-Safety Considerations

NDET provides a multi-threaded architecture and microsecond-clock accuracy on most Portable Operating System Interface (POSIX)-compliant systems. A dedicated control thread tracks emulation time and dispatches discrete event execution to a pool of worker threads as event invocation times occur.

The worker thread pool size is determined by the user when the controller is initialized. Initialization of lightweight processes (aka threads) requires a non-trivial amount of time; consequently, all worker threads are created and initialized prior to initiating the emulation.

The API makes heavy use of conditional variables and mutex mechanisms to provide efficiency and thread safety. The API cannot prevent users from creating race conditions or violating mutual exclusion constraints in their own EEC classes. If users do not share variables or system handles between their EEC classes, there should no thread safety issues for them to manage.

6. Planned Future Capabilities

The NDET system is in its early stages; however, it has already demonstrated its usefulness to the NSRL team controlling an experiment in trust.[3] This section discusses some features planned for future releases.

NDET currently executes in real time, because it was initially designed to support emulation-based experimentation. NSRL researchers are exploring methods to leverage network simulations for experimentation, which will necessitate such a feature in NDET. Because NDET maintains a real-time clock on which event calls are based, this feature will require a decoupling of event execution from the real-time clock.

Another time-based feature is “manual stepping” of the experiment driver. This feature would be most useful in debugging experiments and non-time-based simulations.

Generic resource and store base classes could be constructed to simulate constraining conditions. The generic resource class would provide a standard representation of limited, discrete resources that constrain event flow. A common real-world example would be queues such as traffic lanes or teller windows. Similarly, the generic store would model a standard representation for consumable resources that constrain event flow, such as gallons of fuel or spending power. Stores are distinct from resources in that stores are not returned to an available state after use (e.g., a gallon of fuel does not become available again after one is finished using it).

Collaborations between the NSRL researchers and researchers from other institutions often create a need for an experimentation “mash-up.” Future experiments could benefit from synchronization of multiple distinct experiments, which may be possible using NDET.

The initial version of NDET was developed using C++, which is a common language used in experimentation; however, Python and Java are also prevalent in network science experimentation, so NSRL researchers will explore development of NDET bindings for both of those languages.

Lastly, while initial results indicate performance measurements with microsecond accuracy, jitter can be reduced by dedicating 1 central processing unit (CPU)/core to the controller thread and dispatching worker threads to other CPUs/cores. Performance may also be improved by correcting execution delay using the statistics collected during event execution to correct for consistent deltas between scheduled time and actual invocation times. For instance, if discrete event launches consistently take 5 µs, NDET could compensate by invoking events 5 µs early.

7. Conclusion

NDET has been proven to be a benefit to network science experimentation in NSRL. Further use of this capability and sharing it with collaborators will increase our understanding of its utility and limitations. NSRL researchers will continue to develop NDET and will make it available for public use on the ARL public web site.[4]

8. References

1. Extendable Mobile Ad-hoc Network Emulator (EMANE) [accessed 2015], http://www.nrl.navy.mil/itd/ncs/products/emane.

2. Common Open Research Emulator (CORE) [accessed 2015], http://www.nrl.navy.mil/itd/ncs/products/core.

3. Chan K, Cho JH, Chan K, Trout T, Wampler J, Toth A, Rivera B. trustd: Trust Daemon Experimental Testbed for Network Emulation. MILCOM 2015.

4. NSRL public web site [accessed 2015], http://www.arl.army.mil/nsrl.

List of Symbols, Abbreviations, and Acronyms

API	application programming interface
ARL	US Army Research Laboratory
CORE	Common Open Research Emulator
CPU	central processing unit
DEDT	discrete event definition
DES	discrete event simulation
EEC	event execution class
EMANE	Extendable Mobile Ad hoc Network Emulator
NDET	NSRL Discrete Event Toolkit
NRL	US Naval Research Laboratory
NSRL	Network Science Research Laboratory
POSIX	Portable Operating System Interface
RF	radio frequency
UML	unified modeling language

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIO LL IMAL HRA MAIL & RECORDS MGMT
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
1 (PDF)	US ARMY RESEARCH LAB RDRL CIN A KOTT
3 (PDF)	US ARMY RESEARCH LAB RDRL CIN T S KREPPS A TOTH B RIVERA